Distributed Privacy-Preserving Transparency Logging

Tobias Pulls
Dept. of Mathematics and Computer Science
Karlstad University
Universitetsgatan 2, Karlstad, Sweden
tobias.pulls@kau.se

Roel Peeters Karel Wouters
KU Leuven, ESAT/COSIC & iMinds
Kasteelpark Arenberg 10 bus 2452
3001 Leuven, Belgium
firstname.lastname@esat.kuleuven.be

ABSTRACT

We present a transparency-enhancing tool in the form of a cryptographic scheme that enables data processors to inform users about the actual data processing that takes place on their personal data. Our proposed solution can handle arbitrary processes while offloading storage and interactions with users to dedicated log servers. On top of strong integrity and confidentiality properties, our scheme takes users' privacy one step further by making it impossible to link multiple log entries for the same user or user identifiers across multiple data processors (for distributed processes). Our proposed solution has several applications, e.g., it can make access to electronic health records transparent to the patients to whom the records relate. Furthermore, we are the first to formalise the required security and privacy properties in this setting in a general manner (not specifically for our scheme) and prove that our scheme fulfils these. Finally, we show that our scheme is applicable in practice, providing performance results for a prototype implementation.

Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Protection—*Cryptographic controls*; E.3 [**Data**]: Data Encryption—*Public key cryptosystems*

Keywords

Transparency-enhancing tool, privacy, distributed, applied cryptography, data processing

1. INTRODUCTION

Transparency of data processing is often a requirement for compliance to legislation and/or business requirements: e.g., keeping access logs to electronic health records (EHR), generating bookkeeping records or logging data to black boxes. Furthermore, transparency is recognised as a key privacy principle, e.g., in the EU Data Protection Directive (DPD) 95/46/EC Articles 7, 10, and 11; and in the Swedish Patient Data Act ("Patientdatalagen") SFS (2008:355) that states that patients have the right to see who has accessed their EHR. In general, increased transparency of data processing may increase the end-users' trust in the data processor[1], especially if the data processing is distributed as in cloud computing [12]. Other applications can be found in eGovernment services, where enhanced transparency towards citizens is a key element [22].

Beyond technical considerations of *what* to make transparent to *whom* to accurately represent data processing, there are a number of social and economic issues that need to be taken into account when determining what is an adequate level of transparency. As recognised in recital 41 of the EU DPD, too detailed descriptions of data processing may reveal business-sensitive information of the data processors, such as trade secrets. Furthermore, employees of the data processor may experience the requirement of transparency as a breach of their own privacy [17]. The work presented in this paper is agnostic to what information is made transparent. We focus on ensuring that only the user whose personal data is being processed can read the logged information. This conservative approach ensures the end-users' privacy. In general, sharing information is easier than removing access to information that has already been provided to a party. Our approach has the added benefit of reducing the negative effects of transparency on the employees of the data processor, since their actions are only made transparent towards the user whose data they are processing. E.g., the logs cannot be misused to monitor the employees' performance.

Information about how data will be processed by data processors is usually represented by a *privacy policy* that states what data is requested for which purpose, whether the data will be forwarded to third-parties, how long the data will be retained, and so on. A privacy policy informs the potential user *before* any data is disclosed, so that a user can give *informed consent* to the data processing by the service provider. Our approach enables data processors to inform users about the actual data processing that occurs *after* users have disclosed data, taking the privacy of users into account. Conceptually, access to this information enables a user to *verify* that the actions of the data processor are in line with the privacy policy[2].

WPES'13 2013, November 4, Berlin, Germany
ACM 978-1-4503-2485-4/13/11 ...$15.00.
http://dx.doi.org/10.1145/2517840.2517847.

[1] We opt for the *technical* terminology of data processor and user, as opposed to the EU DPD that uses a more formal terminology (including data controller, data subject, ...).
[2] Assuming that (1) the semantics of the privacy policy and information provided to the user allow reasonable compari-

Our tool allows a data processor to use a set of *log servers*, creating log entries that describe how the user's data has been processed. Users can then, at their own discretion, query these logs to learn of the data processing performed on their disclosed data. Figure 1 illustrates this setting between a user Alice and data processor Bob. Alice discloses some data to Bob under a privacy policy. While Bob is processing Alice's data, Bob generates log entries at a log server that records the processing. Alice can later retrieve the log entries that contain the description of data processing and verify if Bob's actions are in line with his privacy policy.

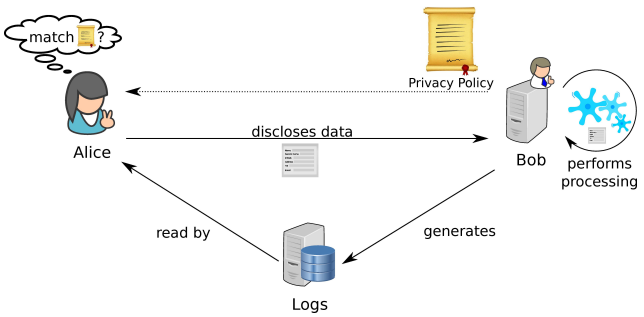

Figure 1: High-level overview of our transparency-enhancing tool's usage.

The simple setting illustrated in Figure 1 is missing one important aspect of data processing: data processing is often distributed. Data concerning users may be shared by the data processor, or other data processors may be used by the initial data processor. We denote the actions, performed by a set of data processors, for a particular task, a *process*. Figure 2 illustrates a setting for a process (solid lines) where Alice discloses data to Bob, the initial data processor. Bob then shares (part of) Alice's data with the downstream data processor Charlie, who in turn shares (part of) Alice's data with data processors Dave and Eve. While data processors Bob, Charlie, Dave, and Eve process Alice's data, all of them continuously log descriptions of their processing (dashed lines) to their, potentially different, log servers. Alice can later reconstruct the *log trail* of the data processing on her data.

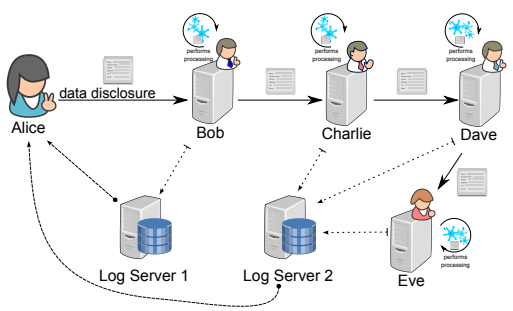

Figure 2: Data processing is often distributed, where data disclosed by a user *Alice* may be shared by the initial data processor *Bob* with other data processors *Charlie*, *Dave*, and *Eve*. Downstream data processing of Alice's data should also be logged.

son, and (2) that the information provided by the data processor is *complete* (see Section 3.1).

In the case of an EHR system, in Figure 2, Alice could be the patient and Bob, Charlie, Dave, and Eve medical journal systems at one or more medical institutions. All access by medical staff to Alice's medical records will be logged such that Alice can later see who has read her medical records.

Because log entries contain descriptions of data processing on personal data, the log entries themselves also contain personal data. Therefore, when implementing such a log system, there are several privacy and security issues that need to be addressed. Returning to the EHR example, knowing who has accessed a person's medical records is sensitive information. In fact, even knowing that a person has been treated at a particular medical institution may be sensitive. In other words, not only the content of the log entries but also the fact that log entries exist for a certain individual is sensitive information. Furthermore, imagine the case of medical personnel illegitimately accessing EHRs. In such a case, the offenders are also likely to attempt to cover the traces of their actions, for example by deleting the generated log entries. Therefore, alterations to log entries need to be detectable once they have been constructed. Similar arguments can be made for the case of business-sensitive information being logged for a company.

Our contributions are as follows: we present a transparency-enhancing tool in the form of a cryptographic scheme for distributed transparency logging of data processing while preserving the privacy of users. Our scheme can handle arbitrary processes while offloading storage and interactions with users to dedicated log servers. Compared to earlier related work, our scheme is the first to define and formally prove several privacy properties in this setting. We also present a performance evaluation of a prototype implementation, showing that our scheme is feasible.

The structure of the remainder of this paper is as follows. Section 2 discusses related work. Section 3 presents our general model and definitions. This is followed in Section 4 by a thorough explanation of our scheme. Section 5 evaluates the scheme's security and privacy properties. We present a performance evaluation of our prototype implementation in Section 6. Finally, Section 7 provides concluding remarks.

2. RELATED WORK

Sackmann *et al.* [18] presented the earliest work on providing transparency of data processing by using cryptographic systems from the secure logging area. Trusted auditors use secure logs of data processing as so called "privacy evidence" to compare the actual processing with the processing stated in a privacy policy that users have consented to. Wouters *et al.* [24] and Hedbom *et al.* [9] tackle privacy issues in a setting with one key difference from the work of Sackmann *et al.*: users take on the primary role of auditors of the logged data that relates to them, arguably removing the need for trusted auditors. This change of setting is the primary source of potential threats to the privacy of users (ignoring the content of log entries), since log entries now relate to different users who are actively participating in the scheme. These threats are primarily due to the *linkability* of entries to users and user identifiers between logs. Wouters *et al.* address the linkability issue between logs on different log servers in a distributed setting, while Hedbom *et al.* address linkability between log entries in one log. The scheme we propose in this paper addresses several issues in prior work, such as truncation attacks (described briefly later) or

partial linkability of log entries. Furthermore, we propose generalised notions of security and privacy in this setting and prove that our scheme fulfils these notions. Finally, we provide implementation results on our scheme.

Note that our work, like [9, 18], builds upon concepts from the secure logging system by Schneier and Kelsey [19]. A thorough review of related work in the secure logging area is given in a technical report by Pulls *et al.* [16] together with a preliminary informal version of the scheme presented in this paper. For this preliminary version, it was shown by Vliegen *et al.* [23] how it could be strengthened by adding trusted hardware. Finally, Peeters *et al.* [15] present a high-level description of our scheme.

3. OUR MODEL

This section first provides an overview of the setting and our adversary model within the setting. Next, we present high-level requirements that are then formalised.

3.1 Setting and Adversary Model

Figure 3 depicts a very simple distributed process. As processing is taking place for Alice's data, Bob continuously sends data describing the data processing to its log server that turns the data into *log entries* for Alice. Each log entry at a log server concerns one user. At some point in time, Bob wishes to *fork* the process to a *downstream* data processor Dave. When a process forks, the user is not involved, yet transparency logging of data processing should continue, potentially to a different log server. We allow the structure of the process to be *dynamic*, meaning that the structure of the process is not necessarily predetermined. When Alice disclosed her data to Bob, neither Alice nor Bob needed to be aware of the fact that Bob would later share Alice's data with Dave. To realise this, we store metadata in the user's log when a fork takes place that enables the user to follow each fork in the process. In this case, when Bob shared the data with Dave, he logged some metadata for Alice. The exact structure of this metadata is later presented in Section 4.5. All of these log entries with metadata, spread across different log servers, form a *log trail* between log servers that only the user can reconstruct.

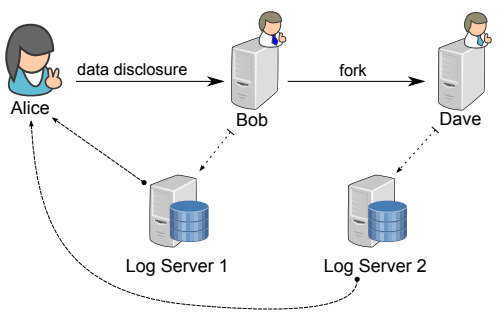

Figure 3: Setting for user *Alice*, data processor *Bob*, and downstream data processor *Dave*.

We consider the properties provided by the scheme for log entries that are *committed prior to compromise* of one or more log servers and/or data processors. The users are fully trusted. In other words, we initially trust all entities in the scheme up to a point in time t when an adversary compromises one or more log servers and/or data processors. This is the standard adversary model within the secure logging area [2, 11, 13, 19, 25]. Log servers keep some *state* information, apart from the storage needed for the log entries themselves. When a log server is compromised, the adversary has full access to its internal state and storage. Our goal is to achieve strong cryptographic properties for all created log entries, and the metadata kept to create them (state), before t.

We argue that assuming initial trust in a data processor is reasonable, since without (some degree of) initial trust, the user will not disclose data to the data processor in the first place. Assuming that the data processor is trustworthy, its automated data processing should be setup to automatically log data processing through our tool. When the data processor becomes compromised, either by an external party or by an insider such as an employee, all descriptions of data processing that have been made up to that point should fulfill the requirements outlined in the following sections. Once compromised, little guarantee can be given about future processing. For log servers, the required initial trust can be minimised by introducing trusted hardware at the log server, e.g., as described by Vliegen *et al.* [23]. We also stress that auditability is an important property of a logging scheme, since the relationship between the log server and the data processor is a fragile one: the data processor trusts the log server to log as instructed, for the correct user, while the log server trusts the data processor to encrypt for the correct user. However, both trusted hardware and auditability are outside the scope of this paper.

For communication, we assume secure channels between all entities. Furthermore, between users and log servers we assume an anonymous channel.

3.2 Requirements

We present four high-level requirements for our scheme and briefly motivate why they are important in our setting. In Section 3.4, these are formally defined and in Section 5 we prove that the scheme presented in Section 4 realises these.

R1 The adversary should not be able to make undetectable modifications to logged data (committed prior to compromise), i.e., the *integrity* of the data should be ensured once stored.

R2 Only the user should be able to read the data logged for him, i.e., the data should be *confidential* (secrecy).

R3 From the log and the state, it should not be possible for an adversary to determine if two log entries (committed prior to compromise) are related to the same user or not, i.e., *user log entries* should be *unlinkable*.

R4 *User identifiers* (setup prior to compromise) across multiple data processors or log servers should be *unlinkable*.

R1 and R2 are the basic notions of forward integrity and secrecy/confidentiality, from the secure logging area, ensuring that logged data cannot be modified or read by an adversary. R3 and R4 emerge primarily as a consequence of our setting, where having multiple *recipients* of logged data leads to the need to protect the *privacy* of the recipient users. R3 protects the privacy of the user by preventing log entries from being linked to a particular user. If log entries could be linked together, patterns on log trails can reveal sensitive information. For example, the number of log entries in a

trail can serve as a unique fingerprint of an event, generating a link to a data processor or a user. R4 removes the links between each step (fork) in the process.

3.3 Notation

We denote the security parameter as $k \in \mathbb{N}$. A function $f : \mathbb{N} \to \mathbb{R}$ is 'polynomial' in k if $f(k) = O(k^n)$, with $n \in \mathbb{N}$. A function is 'negligible' if, for every $c \in \mathbb{N}$ there exists an integer k_c such that $f(k) \leq k^{-c}$ for all $k > k_c$. We denote a negligible function by $\epsilon(k)$. $\texttt{Oracle}(args) \to response$ denotes an algorithm or oracle with input $args$ that outputs $response$. An adversary $\mathcal{A}$ that outputs $response$ after making an arbitrary number of queries (in arbitrary order) to the listed oracles is denoted as $response \leftarrow \mathcal{A}^{\texttt{oracle}_1,\texttt{oracle}_2,\texttt{oracle}_3}()$. $\mathbf{Exp}_{\mathcal{A}}^{property}(k)$ is the experiment (in the security parameter k) that the challenger sets up for an adversary $\mathcal{A}$ against property $property$. $\mathbf{Adv}_{\mathcal{A}}^{property}(k)$ is the advantage of the adversary in breaking property $property$ (for a given security parameter). The advantage is between 0 and 1 for computational problems, between 0 and 1/2 for distinguishable problems[3].

3.4 Model and Definitions

To setup a system, we define two general algorithms:

- $\texttt{GenerateIdentifier}() \to \text{E}$: to generate a random, new and suitable identifier for an entity E within the logging scheme.
- $\texttt{SetupEntity}(\text{L}, \text{E})$: to setup the state at the log server L for entity E.

Without loss of generality, we assume a single data processor P and a single log server L. The data processor has been initialised at the log server, $\texttt{SetupEntity}(\text{L}, \text{P})$ was executed. Users are added dynamically by the adversary. The log server maintains a state for the data processor and every user. The entire log is accessible to the adversary at all times. Let $\mathcal{A}$ denote the adversary that can adaptively control the system through a set of oracles:

- $\texttt{CreateUser}(\lambda) \to \text{U}_i, l$: this oracle calls $\texttt{GenerateIdentifier} \to \text{U}_i$ and runs $\texttt{SetupEntity}(\text{L}, \text{U}_i)$ to setup the state for a new user at the log server. Given the data λ an initial log entry l is created for U_i by calling $\texttt{CreateEntry}(\text{U}_i, \lambda)$. The generated user identifier and log entry are returned.
- $\texttt{CreateEntry}(\text{U}, \lambda) \to l$: this oracle creates a log entry l at log server L for data processor P and user U on the data λ. The log entry is returned.
- $\texttt{CorruptLogServer}() \to \texttt{State}, \#entries$: corrupt the log server L, which returns the entire state of the log server and the number of created log entries before calling this oracle.

Note that, for properties with prefix forward (introduced shortly), the adversary cannot invoke any further queries after it makes a call to the `CorruptLogServer` oracle.

[3]The probability of guessing is already 1/2.

3.4.1 R1: Forward Integrity

We adopt the definitions of Bellare and Yee[6] for forward integrity (FI) and deletion-detection FI secure logging schemes. FI captures that at the time of comprising the log server, no log entries before that time can be forged without being detected. To ensure that no modifications to the log prior to compromise of the log server can be made, truncation attacks[4] also need to be taken into account. For this reason deletion-detection FI is required.

First we define what constitutes a valid log entry: $\texttt{valid}(l, i)$ returns whether or not the full log trail for every user verifies when l_i (the log entry created at the i-th call of `CreateEntry`) is replaced by l. Now, we are ready to present the experiment that the challenger sets up for $\mathcal{A}$ attacking the forward integrity (FI):
$\mathbf{Exp}_{\mathcal{A}}^{FI}(k)$:

1. $l \leftarrow \mathcal{A}^{\texttt{CreateUser},\texttt{CreateEntry},\texttt{CorruptLogServer}}()$
2. Return $\exists\, i \leq \#entries : l \neq l_i \wedge \texttt{valid}(l, i)$.

The advantage of the adversary is defined as

$$\mathbf{Adv}_{\mathcal{A}}^{FI}(k) = Pr\left[\mathbf{Exp}_{\mathcal{A}}^{FI}(k) = 1\right].$$

Definition 1. *A logging system provides computational forward integrity, if and only if for all polynomial time adversaries $\mathcal{A}$, it holds that $\mathbf{Adv}_{\mathcal{A}}^{FI}(k) \leq \epsilon(k)$.*

Definition 2. *A logging system provides computational deletion-detection forward integrity, if and only if it is FI secure and the log verifier can determine, given the output of the log system and the time of compromise, whether any prior log entries have been deleted.*

3.4.2 R2: Secrecy

The adversary $\mathcal{A}$ aiming to compromise the secrecy of the data contained within the log entries (SE), has to guess the bit b, for the modified `CreateEntry` oracle:

- $\texttt{CreateEntry}(\text{U}, \lambda_0, \lambda_1)_b \to l$: this oracle creates a log entry l at log server L for data processor P and user U on the data λ_b. The log entry is returned.

$\mathbf{Exp}_{\mathcal{A}}^{SE}(k)$:

1. $b \in_R \{0, 1\}$
2. $g \leftarrow \mathcal{A}^{\texttt{CreateUser},\texttt{CreateEntry},\texttt{CorruptLogServer}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}_{\mathcal{A}}^{SE}(k) = \frac{1}{2} \cdot \Big| Pr\left[\mathbf{Exp}_{\mathcal{A}}^{SE}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}_{\mathcal{A}}^{SE}(k) = 1 | b = 1\right] - 1 \Big|.$$

Definition 3. *A logging system provides computational secrecy of the data contained within the log entries, if and only if for all polynomial time adversaries $\mathcal{A}$, it holds that $\mathbf{Adv}_{\mathcal{A}}^{SE}(k) \leq \epsilon(k)$.*

[4]A truncation attack is when an attacker deletes a continuous subset of log entries at the end of a log, see e.g. [13].

3.4.3 R3: Forward Unlinkability of User Log Entries

The goal of the adversary $\mathcal{A}$ attacking the forward unlinkability of user log entries (FU) is to guess the bit b, for the modified `CreateEntry` oracle:

- `DrawUser`(U_i,U_j) $\rightarrow$ *vuser*: this oracle generates a virtual user reference, as a monotonic counter, *vuser* and stores ($vuser, \mathrm{U}_i, \mathrm{U}_j$) in a table $\mathcal{D}$. If U_i is already referenced as the left-side user in $\mathcal{D}$ or U_j as the right-side user, then this oracle returns $\perp$ and adds no entry to $\mathcal{D}$. Otherwise, it returns *vuser*.
- `Free`(*vuser*): this oracle removes the triple ($vuser, \mathrm{U}_i, \mathrm{U}_j$) from table $\mathcal{D}$.
- `CreateEntry`($vuser$,λ)$_b$ $\rightarrow l$: from the table $\mathcal{D}$, this oracle retrieves the corresponding ($\mathrm{U}_i, \mathrm{U}_j$). Depending on the value of b, *vuser* either refers to U_i or U_j. A log entry l is created at log server L for data processor P and user U_i (if $b = 0$) or U_j (if $b = 1$) on the data λ. The log entry is returned.

Note that logging a message for a specific user U_i is still possible. This is required for the `CreateUser` oracle, that needs to log a message for the created user. By calling `DrawUser`(U_i,U_i) $\rightarrow$ *vuser*, *vuser* refers to U_i, regardless of the bit b.

$\mathbf{Exp}^{FU}_{\mathcal{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathcal{A}^{\texttt{CreateUser},\texttt{DrawUser},\texttt{Free},\texttt{CreateEntry},\texttt{CorruptLogServer}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{FU}_{\mathcal{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{FU}_{\mathcal{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{FU}_{\mathcal{A}}(k) = 1 | b = 1\right] - 1 \right| .$$

Definition 4. *A logging system provides computational forward unlinkability of user log entries, if and only if for all polynomial time adversaries $\mathcal{A}$, it holds that $\mathbf{Adv}^{FU}_{\mathcal{A}}(k) \leq \epsilon(k)$.*

3.4.4 R4: Unlinkability of User Identifiers

This property implies multiple data processors, hence the oracles `CreateUser` and `CreateEntry` need to be made data processor dependent. Without loss of generality, we can still assume that there is a single log server and that all data processors have been initialised at this log server. An attacker against Unlinkability of User Identifiers (UU) has to guess the bit b, used in the `Fork` oracle:

- `Fork`($\mathrm{U}, \mathrm{P}_0, \mathrm{P}_1$)$_b$ $\rightarrow \mathrm{U}', l$: for a user U setup with data processor P_0, this oracle constructs U' either by randomising U($b = 0$) or by generating a new identifier by calling `GenerateIdentifier`() ($b = 1$). Next, U' is setup for P_1 at log server L. The resulting data, together with the data used to randomise the identifier ($b = 0$) or random data of equal length ($b = 1$), is logged for user U and data processor P_0. The new user and log entry is returned.

$\mathbf{Exp}^{UU}_{\mathcal{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathcal{A}^{\texttt{CreateUser},\texttt{CreateEntry},\texttt{Fork},\texttt{CorruptLogServer}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{UU}_{\mathcal{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{UU}_{\mathcal{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{UU}_{\mathcal{A}}(k) = 1 | b = 1\right] - 1 \right| .$$

Definition 5. *A logging system provides computational unlinkability of user identifiers, if and only if for all polynomial time adversaries $\mathcal{A}$, it holds that $\mathbf{Adv}^{UU}_{\mathcal{A}}(k) \leq \epsilon(k)$.*

4. OUR SCHEME

This section describes our proposed scheme. First, we explain the idea behind the log entry structure and the state kept at the log server. Then, we introduce the required cryptographic building blocks. Finally, we describe the components of our proposal in detail.

4.1 Log Entry Structure and State

A log entry is created by a log server for a user U when the data processor P sends some data to log. A log entry consists of five fields:

Data The data field contains the actual data to be logged in an encrypted form, such that only the user can derive the plaintext.

$\mathbf{IC_U}$ The *index chain* field for the user serves as an identifier for the log entry for the user. The values of this field create a chain that links all log entries for the user together. Only the user can reconstruct this chain.

$\mathbf{DC_U}$ The *data chain* field for the user allows the user to verify the validity of this log entry. All entries that were created for this user are chained together, leading to *cumulative verification*, i.e., verifying the integrity of one log entry verifies all the previous log entries too.

$\mathbf{IC_P}$ The index chain field for the data processor.

$\mathbf{DC_P}$ The data chain for the data processor.

The data processor supplies the value that goes into the data field, while the index chain fields are derived from the state kept by the log server. The data chain fields are derived from the state kept by the log server, the data field and index fields from the log entry. A log server keeps the following values in its state *for each* user and data processor:

AK The current authentication key, used as a key when generating the DC field of the next log entry, and when updating state after log entry creation.

IC An intermediate index chain value, used to generate the next IC field of a log entry, derived from the IC field of the previous log entry for this entity.

DC An intermediate data chain value, used to generate the next DC field of a log entry, derived from the DC field of the previous log entry for this entity.

The state is *updated* each time a log entry is created, making it hard to recover the previous values stored in state. This is the mechanism at the core of the *prior to compromise* adversary model in our setting. Due to the fact that the state kept by log servers is continuously updated as log entries are created, the adversary is unable to reconstruct prior states needed for manipulating log entries created prior to compromise. Figure 4 illustrates the interplay between log entries, the log server's state, and our adversary model.

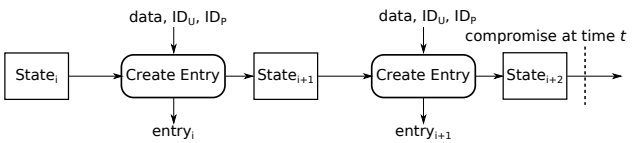

Figure 4: The interplay between log entries, the log server's state, and our adversary model.

4.2 Cryptographic Building Blocks

To provide forward integrity and forward unlinkability of user identifiers, we make use of a cryptographic hash function $\texttt{H}(\cdot)$ and a cryptographic message authentication code generation algorithm $\texttt{MAC}_K(\cdot)$ under a secret key K. Each time a log entry is created the secret key K used for the MAC algorithm is evolved using the hash function. For the MAC algorithm, we make use of the general HMAC construction.

In our scheme, the identifiers of entities are public keys that can be used to encrypt messages for this entity. An encryption scheme $\prod$ consists of three algorithms: `GenerateKey()`, $\texttt{Enc}_{\text{PK}}(p)$, and $\texttt{Dec}_{\text{SK}}(c)$. `GenerateKey()` generates a new random keypair (PK, SK) consisting of public key PK and private key SK. $\texttt{Enc}_{\text{PK}}(p)$ outputs the encryption of a plaintext p with public key PK. $\texttt{Dec}_{\text{SK}}(c)$ for a keypair (PK, SK), outputs the plaintext p by decrypting the ciphertext $c \leftarrow \texttt{Enc}_{\text{PK}}(p)$. For our logging scheme to provide secrecy, $\prod$ should provide indistinguishability under chosen-plaintext attack (IND-CPA) security [4]. Furthermore, for our logging scheme to achieve forward unlinkability of user identifiers, $\prod$ should also be key private under chosen plaintext attack (IK-CPA) [3]. For efficiency, a hybrid cipher with asymmetric key encapsulation and symmetric data encapsulation mechanism (KEM-DEM) [10] will be used. We selected the elliptic curve integrated encryption scheme (ECIES) [21], as defined in ISO/IEC 18033-2. User identifiers are generated as follows: `GenerateKey()`$\rightarrow (\text{PK}, \text{SK})$, where $\text{SK} = x \in_R \mathbb{Z}_n$ and $\text{PK} = X = xG$, for a generator G of the curve.

The data processor and log server have separate key pairs used for signing purposes, in order to establish the origin of messages. A signature scheme consists of three algorithms: `GenerateKey()`, $\texttt{Sign}_{\text{SK}}(m)$, and $\texttt{Verify}_{\text{PK}}(\sigma, m)$. `GenerateKey()` generates a random, new keypair (PK, SK). $\texttt{Sign}_{\text{SK}}(m)$ outputs a signature σ on message m using private key SK. $\texttt{Verify}_{\text{PK}}(\sigma, m)$ verifies that σ is a valid signature on m using the public key PK, and returns true if the signature is valid, otherwise false. The signature scheme should be non-repudiable and selectively unforgeable under known message attack [8]. We selected ECDSA [14].

4.3 Setup and Stopping

Before a data processor can start logging data for users, it needs to be set up at its log server. First the data processor generates a key pair and uses the public key as its identifier at its log server. The log server runs Algorithm 1 with the data processor's public key as input and returns the output to the data processor. This algorithm initialises the log server's state for the provided identifier and outputs the initial *authentication key* (AK_0) encrypted under this identifier. To guarantee the origin of the initial authentication key, it is signed by the log server prior to encryption. The data processor decrypts the output of the log server, verifies the signature and stores AK_0.

Algorithm 1 Setup of an entity at a log server.

Require: An entity identifier ID_E.
Ensure: The encrypted initial values in `State`.
1: $AK_0 \leftarrow \texttt{Rand}(|H(\cdot)|)$
2: $\texttt{State}(\text{ID}_\text{E}, IC) \leftarrow \texttt{MAC}_{AK_0}(\text{ID}_\text{E})$
3: $\texttt{State}(\text{ID}_\text{E}, DC) \leftarrow \text{null}$
4: $\texttt{State}(\text{ID}_\text{E}, AK) \leftarrow H(AK_0)$
5: **return** $\texttt{Enc}_{\text{ID}_\text{E}}\left(AK_0, \texttt{Sign}_{\text{SK}_\text{L}}(AK_0, \text{ID}_\text{E})\right)$

To set up users, the user first generates a fresh user identifier as follows:

$$\text{ID}_\text{U} \leftarrow \text{PK}\,, \text{with}\,(\text{PK}, \text{SK}) \leftarrow \texttt{GenerateKey()} \qquad (1)$$

Then, the user runs Protocol 1 to set up transparency logging at the data processor. At the end of the protocol, the user will be in possession of the initial authentication key, and be assured that it was generated by the log server. This initial authentication key will be used by the user to download and verify all log entries related to him from the log server, as described later in Section 4.6. The data processor keeps track of the association between the data that the user disclosed and the user identifier ID_U such that the data processor can later log data processing for the correct user.

Protocol 1 Setup between user U and data processor P with log server L.

Require: P is set up at L, user identifier ID_U.
Ensure: User knowns AK_0 for ID_U.
1: $\text{U} \longrightarrow \text{P} \longrightarrow \text{L} : \text{ID}_\text{U}$
2: $\text{L} : \alpha \leftarrow \texttt{Enc}_{\text{ID}_\text{U}}\left(AK_0, \texttt{Sign}_{\text{SK}_\text{L}}(AK_0, \text{ID}_\text{U})\right) \leftarrow$ Algorithm 1 with input ID_U
3: $\text{L} \longrightarrow \text{P} : \alpha$
4: $\text{P} \longrightarrow \text{U} : \alpha, \texttt{Sign}_{\text{SK}_\text{P}}(\alpha)$

When a data processor has finished processing on a user's data, the data processor constructs one final log entry for the user with the marker M_S that signals to the user that the data processor has finished its processing. We describe how to create a log entry in Section 4.4. Next, the data processor instructs the log server to delete the state kept for that user from the log server.

4.4 Generating Log Entries

When a data processor performs processing on a user's disclosed data, it logs a description of the processing to the log trail of the user located at the log server used by the data processor as described in Protocol 2. The data processor first signs the data to log (to prove the origin to the user) and then encrypts the data and signature under the public key of the user. Next, the data processor sends the resulting ciphertext to the log server who creates a log entry.

Protocol 2 The generate log entry protocol for data processor P with log server L.

Require: The processor identifier $\mathrm{ID_P}$, user identifier $\mathrm{ID_U}$, and data λ to log.
Ensure: Log entry l created for $\mathrm{ID_U}$ and $\mathrm{ID_P}$ with data λ.

1: P: $d \leftarrow \mathtt{Enc}_{\mathrm{ID_U}}\Big(\lambda, \mathtt{Sign}_{\mathrm{SK_P}}(\lambda)\Big)$
2: P $\longrightarrow$ L : $\mathrm{ID_P}, \mathrm{ID_U}, d$
3: L $\longrightarrow$ P : $l \leftarrow$ Algorithm 2 with input $\mathrm{ID_P}, \mathrm{ID_U}$ and d

The log server creates the log entry by running Algorithm 2. First, the log entry is created (steps 1–5) and written into storage (step 6). Next, the log server's state is updated for both the involved user and data processor (steps 7–12). In the process of updating the state, any old values are overwritten and no longer accessible afterwards. At the end (step 13), the generated log entry is returned.

Algorithm 2 The algorithm for creating a log entry.

Require: Identifiers for a data processor $\mathrm{ID_P}$ and a user $\mathrm{ID_U}$, and the data d to log.
Ensure: Log entry created and stored.

1: $IC_{\mathrm{U}} \leftarrow \mathtt{H}\Big(\mathtt{State}(ID_{\mathrm{U}}, IC)\Big)$
2: $DC_{\mathrm{U}} \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}}, AK)}\Big(\mathtt{State}(ID_{\mathrm{U}}, DC), IC_{\mathrm{U}}, d\Big)$
3: $IC_{\mathrm{P}} \leftarrow \mathtt{H}\Big(\mathtt{State}(ID_{\mathrm{P}}, IC)\Big)$
4: $DC_{\mathrm{P}} \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{P}}, AK)}\Big(\mathtt{State}(ID_{\mathrm{P}}, DC), IC_{\mathrm{P}}, DC_{\mathrm{U}}, IC_{\mathrm{U}}, d\Big)$
5: $l_i \leftarrow (IC_{\mathrm{U}}, DC_{\mathrm{U}}, IC_{\mathrm{P}}, DC_{\mathrm{P}}, d)$
6: $\mathtt{Storage}(IC_{\mathrm{U}}) \leftarrow \mathtt{Storage}(IC_{\mathrm{P}}) \leftarrow l_i$
7: $\mathtt{State}(ID_{\mathrm{U}}, IC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}}, AK)}(IC_{\mathrm{U}})$
8: $\mathtt{State}(ID_{\mathrm{U}}, DC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}}, AK)}(DC_{\mathrm{U}})$
9: $\mathtt{State}(ID_{\mathrm{U}}, AK) \leftarrow \mathtt{H}\Big(\mathtt{State}(ID_{\mathrm{U}}, AK)\Big)$
10: $\mathtt{State}(ID_{\mathrm{P}}, IC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{P}}, AK)}(IC_{\mathrm{P}})$
11: $\mathtt{State}(ID_{\mathrm{P}}, DC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{P}}, AK)}(DC_{\mathrm{P}})$
12: $\mathtt{State}(ID_{\mathrm{P}}, AK) \leftarrow \mathtt{H}\Big(\mathtt{State}(ID_{\mathrm{P}}, AK)\Big)$
13: **return** l_i

4.5 Forking a Process

When a data processor wishes to involve another data processor in the processing of a user's data, the transparency logging needs to fork such that the data processing can be made transparent to the user at the additional data processor. As part of forking, the public key used to identify the user needs to be *blinded*, to prevent linking the transparency logging on both data processors for the same user. Blinding a public key X is done by generating a random integer $r \neq 0$ in $\mathbb{Z}_n$, and used to generate a new public key X' as follows:

$$X' = X + rG = (x + r)G \tag{2}$$

The corresponding private key is $x + r$, which is easy to compute when knowing both the original secret key x and the blinding factor r.

Protocol 3 describes forking between two data processors, the initiating (that wishes to fork the process) P_a and the receiving P_b, that use log servers L_a and L_b respectively, where L_a and L_b may or may not be the same log server. First, P_a creates a blinded user identifier for the user using Equation (2) (step 1). Next, this new user is set up with data processor P_b and log server L_b, where P_a takes on the role of the user in Protocol 1 (step 2). Finally (step 3), P_a creates a special log entry at its log server L_a for ID_{U} with M_f, a forking marker indicating that a fork has taken place, r, the information needed to generate the corresponding blinded private key, and β, the encrypted initial authentication key from the log server used by the receiving data processor. The user ID_{U} can later, with (M_f, r, β), reconstruct the log trail for ID'_{U} used by data processor P_b at log server L_b.

Protocol 3 The forking protocol between data processors P_a and P_b with log servers L_a and L_b, respectively.

Require: The user identifier $\mathrm{ID_U}$.
Ensure: Forking information written to the log.

1: $\mathrm{P}_a : (\mathrm{ID}'_{\mathrm{U}}, r) \leftarrow$ Equation (2) with input $\mathrm{ID_U}$
2: $\mathrm{P}_a : \beta \leftarrow$ Protocol 1 with data processor P_b and its log server L_b for user identifier $\mathrm{ID}'_{\mathrm{U}}$
3: P_a : run Protocol 2 with L_a, for $\mathrm{ID_U}$ with data $\lambda = (M_f, r, \beta)$

4.6 Reconstruction

When the user disclosed data to the data processor, the user initiated Protocol 1, generating an identifier $\mathrm{ID_U}$ in the process, and getting an initial authentication key AK_0 from the data processor. To reconstruct the log trail, the user first downloads all log entries, stored at the log server used by this data processor, linked to his identifier $\mathrm{ID_U}$. Next, the user validates the downloaded log trail.

Based on how a user is set up at the log server (Algorithm 1), and how log entries are created (Algorithm 2), the following equations describe how the user can generate the index chain value of all the user's log entries, for the i:th log entry where $i \geq 1$, given AK_0 and $\mathrm{ID_U}$:

$$IC_{\mathrm{U}_1} = H\Big(MAC_{AK_0}(ID_{\mathrm{U}})\Big) \tag{3a}$$

$$AK_i = H(AK_{i-1}) \tag{3b}$$

$$IC_{\mathrm{U}_i} = H\Big(MAC_{AK_{i-1}}(IC_{\mathrm{U}_{i-1}})\Big) \tag{3c}$$

A log server will provide any log entry it stores that matches a provided index chain value, so Equation (3) enables users to download all of their log entries at a log server sequentially until the response from the log server no longer contains any data. The straightforward way for a user to sequentially download log entries from a log server may reveal (i) the order in which log entries were generated, and (ii) that log entries belong to the same user, i.e., one can link log entries together. Please note that user behaviour is not considered in our model since the negatively effects to the unlinkability properties of log entries also depend on the setting our system is deployed in. Users can however limit these effects by (i) randomising the order in which log entries are requested, at the cost of introducing some requests for log entries not yet created and (ii) waiting some time between each request for a log entry. Furthermore, users can cache already downloaded log entries and it is assumed that log servers will be serving multiple data processors and users at the same time.

Protocol 4 describes the necessary steps to verify the authenticity of the log trail. First, the user verifies locally the authenticity of the downloaded log entries (steps 1–9.) For each entry, the MAC in the DC_{U} field of the entry is compared to the correct computed value (step 4), and the data field of the entry is decrypted and then the signature of the processor on the logged data is verified (steps 6–8). Next, the user requests the IC_{U} value stored in the log server's

state for $\mathrm{ID_U}$ (steps 10–14). The log server replies by sending the IC_U value (or, if it does not exist, indicating that logging has stopped, a random value of equal length) encrypted under the provided identifier. By sending this value encrypted, only the legitimate user will be able to learn the current IC_U value at the log server. This value is checked against the last entry in the log trail (steps 15–18). This interaction with the log server allows the user to detect deletion of log entries generated for his identifier (prior to compromise of the log server).

Protocol 4 Verify the authenticity of a log trail at log server L for user $\mathrm{ID_U}$.

Require: The user's identifier $\mathrm{ID_U}$, corresponding private key $\mathrm{SK_U}$, initial authentication key AK_0, list of log entries *log*, and the data processor's public key $\mathrm{PK_P}$ for signature verification.
Ensure: True if the log trail in *log* is valid, false otherwise.
1: $index = 0$, $key \leftarrow AK_0$, $chain \leftarrow$ null
2: **while** $index < |log|$ **do**
3: $\quad e \leftarrow log[index], key \leftarrow \mathtt{H}(key)$
4: $\quad$ **if** $e.DC_\mathrm{U} \neq \mathtt{MAC}_{key}(chain, e.IC_\mathrm{U}, e.Data)$ **then**
5: $\quad\quad$ **return** false
6: $\quad (m, \sigma = \mathtt{Sign}_{\mathrm{SK_P}}(m)) \leftarrow \mathtt{Dec}_{\mathrm{SK_U}}(e.Data)$
7: $\quad$ **if** $\mathtt{Verify}_{\mathrm{PK_P}}(\sigma, m) \neq$ true **then**
8: $\quad\quad$ **return** false
9: $\quad chain \leftarrow \mathtt{MAC}_{key}(e.DC_\mathrm{U})$, $index++$
10: $\mathrm{U} \longrightarrow \mathrm{L} : \mathrm{ID_U}$
11: **if** $\mathrm{L} : \mathtt{State}(\mathrm{ID_U}, IC) \neq$ null **then**
12: $\quad \mathrm{L} \longrightarrow \mathrm{U} : r \leftarrow \mathtt{Enc}_{\mathrm{ID_U}}\left(\mathtt{State}(\mathrm{ID_U}, IC)\right)$
13: **else**
14: $\quad \mathrm{L} \longrightarrow \mathrm{U} : r \leftarrow \mathtt{Enc}_{\mathrm{ID_U}}\left(\mathtt{Rand}(|\mathtt{MAC}(\cdot)|)\right)$
15: **if** m contains marker M_s **then**
16: $\quad$ **return** $\mathtt{Dec}_{\mathrm{SK_U}}(r) \neq \mathtt{MAC}_{key}(e.IC_\mathrm{U})$
17: **else**
18: $\quad$ **return** $\mathtt{Dec}_{\mathrm{SK_U}}(r) \stackrel{?}{=} \mathtt{MAC}_{key}(e.IC_\mathrm{U})$

After verification of the log trail, the user searches for any entries with the forking marker M_f. For each such entry, the user uses his private key SK and together with the blinding factor r from the log entry to construct the blinded private key SK′ as described in Section 4.5. With the blinded private key, the user can decrypt the additional payload of the log entry (see Protocol 3) and recover the initial authentication key AK_0' for the forked process. With this information, the user repeats the procedure outlined in this section to reconstruct the rest of the log trail.

Note that a data processor could also retrieve all its created log entries from its log server using a similar approach as described in this section, and verify the integrity of all data by computing the data chain for the data processor.

5. EVALUATION

In this section, we evaluate our scheme with respect to the security and privacy properties defined in Section 3.4.

5.1 R1: Forward Integrity

THEOREM 1. *The proposed logging scheme provides computational forward integrity, according to Definition 1.*

PROOF. This follows directly from the proof of Theorem 3.2 by Bellare *et al.* [5]. They showed that for a standard message authentication scheme and a forward secure pseudo-random generator, the general construction of a key-evolving message authentication scheme is forward secure. □

THEOREM 2. *The proposed logging scheme provides computational deletion-detection forward integrity, according to Definition 2.*

PROOF. First, the scheme provides computational forward integrity. Second, we will show that user can detect if log entries in his trail are deleted. We have to differentiate between two cases: (1) the logging for this user has ended and (2) the logging for this user is still ongoing. In the first case the user will be able to validate its entire log trail until the last entry containing a stopping marker. In absence of this marker in the last log entry, the user will assume to be in the second case and ask the log server for the current $\mathtt{State}(\mathrm{ID_U}, IC)$ value. This value is encrypted under the user's public key. Hence, this mechanism only allows the legitimate user to learn this value. Now we need to show that a corrupted log server cannot produce a previous $\mathtt{State}(\mathrm{ID_U}, IC)$ value, given the current state and all log entries. For an adversary to truncate the log entries for a certain user U given the last entry for this user to retain, $IC_{\mathrm{U}_{last}}$, and the first entry for this user after that, $IC_{\mathrm{U}_{truncated}}$, this adversary has to come up with $\mathtt{State}_{last}(\mathrm{ID_U}, IC)$ that follows $IC_{\mathrm{U}_{last}}$. The following relations exist:

- $\mathtt{State}_{last}(\mathrm{ID_U}, IC) = \mathtt{MAC}_{\mathtt{State}_{last}(ID_\mathrm{U}, AK)}(IC_{\mathrm{U}_{last}})$ and
- $IC_{\mathrm{U}_{truncated}} = H(\mathtt{State}_{last}(\mathrm{ID_U}, IC))$.

Given the one-way nature of the hash function and non-malleability of the MAC algorithm, the advantage of an adversary for creating a valid $\mathtt{State}_{last}(\mathrm{ID_U}, IC)$ is negligible. □

5.2 R2: Secrecy

THEOREM 3. *Under the Decisional Diffie-Hellman (DDH) assumption, the proposed logging scheme provides computational secrecy of the data contained within the log entries, according to Definition 3.*

PROOF. The data field in the log entry, for a user U and data λ_b, is $d = \mathtt{Enc}_{\mathrm{ID_U}}\left(\lambda_b, \mathtt{Sign}_{\mathrm{SK_P}}(\lambda_b)\right)$. For an adversary to distinguish between λ_0 and λ_1 being logged, another adversary with the same success probability can be constructed, breaking the IND-CPA security of the encryption scheme Enc. For the used ECIES encryption scheme, this probability is negligible under the DDH assumption. □

5.3 R3: Forward Unlinkability of User Log Entries

THEOREM 4. *In the random oracle model, under the DDH assumption, the proposed logging scheme provides computational forward unlinkability of user log entries, according to Definition 4.*

We only provide a sketch of the proof. The full proof is given in Appendix A.

PROOF SKETCH. A game-based proof technique proposed by Shoup [20] is used. First, the encryption of the data λ and the signature on it by the data processor for a specific user is replaced by an encryption of this data and signature under a randomised public key. This is possible, given that the encryption scheme provides key privacy under chosen plaintext attack (IK-CPA) [3]. For the used ECIES encryption scheme, the advantage of a distinguisher is equal to the DDH advantage. Second, all instantiations of the hash

function (also within the MAC algorithm since we make use of HMAC) are replaced by a random oracle. Finally, we replace everything by true random values. The obtained advantage of the adversary is smaller than the sum of the DDH advantage and the birthday paradox bound, which is negligible. □

5.4 R4: Unlinkability of User Identifiers

THEOREM 5. *Under the DDH assumption, the proposed logging scheme provides computational forward unlinkability of user identifiers, according to Definition 5.*

We only provide a sketch of the proof. The full proof is given in Appendix B.

PROOF SKETCH. First, we show that the distribution of the randomised user identifiers is equal to the distribution of user identifiers. Second, we will show by reduction that there exists no adversary against the unlinkability of user identifiers under the DDH assumption. □

6. EFFICIENCY

We implemented a prototype of our scheme in the programming language Go [1], with the cryptographic primitives ECIES and ECDSA on NIST P-256, SHA-256, and HMAC using SHA-256. All benchmarks are performed on a laptop running a GNU/Linux based x64 OS with an Intel i5 (quad core 2.6GHz) CPU and 7.7 GB DDR3 RAM. Table 1 provides a benchmark of the algorithms and protocols that make up our scheme, done by Go's built-in benchmarking functionality, which will run a test until it is "timed reliably". The benchmark shows that operations related to encryption and signatures are the main bottlenecks. As a consequence, data processors perform the bulk of the work. Decryption and verification for users are relatively costly. However, in practice, downloading the log entries over an anonymous channel (such as that provided by Tor [7]) possibly with an imposed waiting time, as discussed in Section 4.6, will most likely cause the bottleneck. For log servers, creating log entries is fast, while setup (also part of forking) is costly. Presumably, setup will be relatively infrequent.

Table 1: Benchmark of algorithms and protocols.

Algorithm/Protocol	Time [ms]	Comments
Algorithm 1: Setup Entity	14,8	
Protocol 1: Setup U, L, P	25,0	
Algorithm 2: Create Log Entry	0,1	1 KiB data
Protocol 2: Generate Log Entry	15,0	1 KiB data
Protocol 3: Forking	45,3	
Protocol 4: Verify	180,2	10 entries of 1 KiB data

To get a better idea of how our scheme would perform in practice as a deployed system, we extended our implementation. First, we transformed the data processor to a standalone service (similar to a Syslog server) to which other systems at the data processor send messages that should be logged. The data processor and log server provide RESTful APIs over HTTPS. TLS is provided with the cipher suite ECDHE/RSA/AES/256/CBC/SHA, where the 2048-bit RSA keys are pre-shared. Next, we introduced the concept of *transactions*, analogous to transactions in relational databases. At a data processor, starting a transaction creates a new buffer for user-message pairs. A transaction can then be committed, which takes all user-message pairs in the buffer and creates log entries of them as described in our scheme. At a log server, a transaction buffer works in a similar way: a data processor can create a buffer of index-user-message triplets that when committed gets added as log entries at the log server, in the order of their indices. This index is needed because our scheme requires that the order of log entries is preserved. In return, this means that the data processor can easily send index-user-message triplets in parallel to the log server. For the transaction buffers at a data processor we also added support for parallelism. When a data processor has received a user-message pair, the processor spawns a new Go routine (similar to a lightweight thread) that performs the signing and encryption of the message in the background. This allows the data processor to quickly let other systems at the data processor return to their primary work of data processing, and the data processor service can perform the heavy computations while waiting for the transaction to be committed.

Figure 5 shows the average time (in milliseconds) for generating a log entry depending on the number of entries in each transaction and the size of the data to be logged for each entry. Each entry was logged for a random user that was set up before the start of the experiment. The log server, the data processor, and the application calling the data processor API were run both locally (L) and remotely (R). The local experiment was run on the above described setup. For the remote experiment, the log server was run at Amazon and the data processor in a private cloud at Karlstad University in Sweden. The application calling the data processor API was run on the same laptop as used previously connected to the same university network as the data processor. The Amazon EC2 instance was an M1 Medium instance with 2 ECUs (EC2 Compute Unit), 1 core, and 3.7 GiB of memory. It was hosted in the eu-west-1c zone and ran Ubuntu GNU/Linux 12.10 x64. The private cloud instance was a VMware Ubuntu GNU/Linux 12.04 x64 instance with access to a quad core Intel Xeon E5540 CPU and 4 GiB of memory. We used the Linux *ping* command to measure the latency between 1) the data processor and the Amazon EC2 instance: on average 45.7 ms with a standard deviation of 0.3 ms, and 2) the laptop and the data processor: on average 1.19 ms with a standard deviation of 0.09 ms.

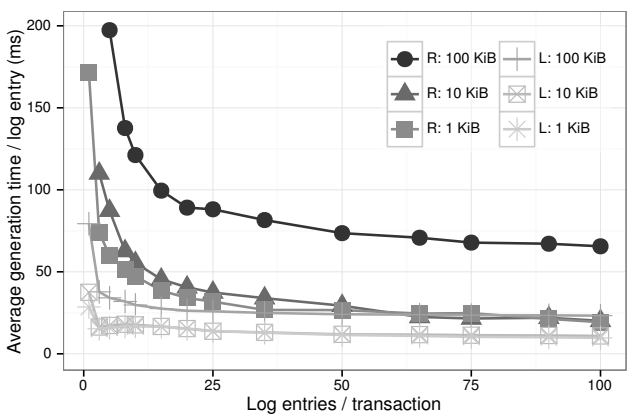

Figure 5: The average time (ms), in a local (L) and remote (R) setting, for generating a log entry depending on the number of log entries per transaction and the size of the data to be logged.

The experiment in Figure 5 clearly shows that a modest transaction size significantly lowers the average log entry generation time. At around 20 entries per transaction, the benefit of adding more entries to a transaction diminishes. The average log entry time does not scale linearly with the size of the logged data. This is mainly due to the fact that the data to be logged is first signed and then encrypted before being sent to the log server. The relative overhead for the signature and public-key encryption is larger for smaller log entries. In the local setting, the difference between generating a log entry with 1 KiB and 100 KiB of data is roughly 13 ms at 50 entries per transaction, resulting in about double the average log entry generation time for logging 100 times more data. In the remote setting, logging 100 KiB of data per entry takes roughly 3 times as long as logging 1 KiB of data. We suspect the increased time in the remote setting is due to the increased latency, but further experiments are needed. Table 2 shows the *goodput*, the throughput measured with respect to the data to be logged, for both the local and remote setting at 100 log entries per transaction.

Table 2: Goodput at 100 log entries per transaction.

Data per log entry	1 KiB	10 KiB	100 KiB
Local setting	87 KiB/s	842 KiB/s	4149 KiB/s
Remote setting	52 KiB/s	497 KiB/s	1525 KiB/s

7. CONCLUSIONS

We introduced a new privacy-preserving transparency-enhancing tool. Our tool generates a log trail for a user, typically the subject of the process that is logged. Dynamic and distributed processes can be logged, and the logging itself can also be distributed across several log servers. The log entries are world-readable, but the strong cryptographic properties of the underlying scheme ensure confidentiality and unlinkability in a broad sense. Moreover, we are the first to formalise these properties in a general way, enabling us to prove that our scheme meets the necessary requirements. We also implemented our scheme in a robust prototype implementation and the first timing results show that the tool can be used in practice.

In the setting of our work, having some initial trust in data processors is unavoidable. However, log servers should not initially have to be trusted. This can be realised in part by introducing *trusted hardware* at the log server. Interesting future work is to improve on the design of such a hardware extension, based upon e.g. the work of Vliegen *et al.* [23], for our scheme. Another venue for future work is to investigate suitable faster encryption and signature primitives, since those operations account for the majority of the computation as shown in Section 6.

Acknowledgments

Tobias Pulls has received funding from the Seventh Framework Programme for Research of the European Community under grant agreement no. 317550. This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007).

8. REFERENCES

[1] The Go Programming Language, version 1.1 RC2.
[2] R. Accorsi. BBox: A Distributed Secure Log Architecture. In J. Camenisch and L. Costas, editors, *EuroPKI '10*, volume 6711 of *LNCS*. Springer, 2010.
[3] M. Bellare, A. Boldyrevay, A. Desaiz, and D. Pointcheval. Key-Privacy in Public-Key Encryption. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT '01*, volume 2248 of *LNCS*, pages 566–582. Springer, 2001.
[4] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In H. Krawczyk, editor, *Advance in Cryptology – CRYPTO '98*, volume 1462 of *LNCS*, pages 26–45. Springer, 1998.
[5] M. Bellare and B. Yee. Forward-Security in Private-Key Cryptography. In M. Joye, editor, *CT-RSA '03*, volume 2612 of *LNCS*, pages 1–18. Springer, 2003.
[6] M. Bellare and B. S. Yee. Forward Integrity For Secure Audit Logs. Technical report, 1997.
[7] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium '04*, pages 303–320. USENIX, 2004.
[8] S. Goldwasser, S. Micali, and R. L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, 17:281–308, 1988.
[9] H. Hedbom, T. Pulls, P. Hjärtquist, and A. Lavén. Adding Secure Transparency Logging to the PRIME Core. In *Privacy and Identity Management for Life*, volume 320, pages 299–314. Springer, 2010.
[10] J. Herranz, D. Hofheinz, and E. Kiltz. KEM/DEM: Necessary and Sufficient Conditions for Secure Hybrid Encryption. Cryptology ePrint Archive, Report 2006/265, 2006.
[11] J. E. Holt. Logcrypt: Forward Security and Public Verification for Secure Audit Logs. In R. Buyya, T. Ma, R. Safavi-Naini, C. Steketee, and W. Susilo, editors, *ACSW Frontiers '06*, volume 54 of *CRPIT*, pages 203–211. Australian Computer Society, 2006.
[12] R. K. L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B.-S. Lee. TrustCloud: A Framework for Accountability and Trust in Cloud Computing. In *SERVICES*, pages 584–588. IEEE Computer Society, 2011.
[13] D. Ma and G. Tsudik. A New Approach to Secure Logging. In V. Atluri, editor, *DBSec*, volume 5094 of *LNCS*, pages 48–63. Springer, 2008.
[14] NIST. FIPS 186-3: Digital Signature Standard (DSS), 2009.
[15] R. Peeters, T. Pulls, and K. Wouters. Enhancing transparency with distributed privacy-preserving logging. ISSE, 2013. To appear.
[16] T. Pulls, K. Wouters, J. Vliegen, and C. Grahn. Distributed Privacy-Preserving Log Trails. Technical Report 2012:24, Karlstad University, Department of Computer Science, 2012.
[17] J. Roberts. No one is perfect: The limits of transparency and an ethic for 'intelligent' accountability. *Accounting, Organizations and Society*, 34(8):957–970, 2009.
[18] S. Sackmann, J. Strüker, and R. Accorsi. Personalization in Privacy-Aware Highly Dynamic Systems. *Communications of the ACM*, 49(9):32–38, 2006.
[19] B. Schneier and J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *USENIX Security Symposium '98*, pages 53–62. USENIX, 1998.
[20] V. Shoup. Sequences of Games: a Tool for Taming Complexity in Security Proofs. Cryptology ePrint Archive, Report 2004/332, 2004.
[21] N. P. Smart. The Exact Security of ECIES in the Generic Group Model. In B. Honary, editor, *Cryptography and Coding*, volume 2260 of *LNCS*, pages 73–84. Springer, 2001.
[22] United Nations Department of Economic and Social Affairs. UN e-Government Survey 2012. E-Government for the People. Technical report, 2012.
[23] J. Vliegen, K. Wouters, C. Grahn, and T. Pulls. Hardware strengthening a distributed logging scheme. In *DSD*, pages 171–176. IEEE, 2012.
[24] K. Wouters, K. Simoens, D. Lathouwers, and B. Preneel. Secure and Privacy-Friendly Logging for eGovernment Services. In *ARES*, pages 1091–1096. IEEE Computer Society, 2008.
[25] A. A. Yavuz and P. Ning. BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems. In *ACSAC*, pages 219–228. IEEE Computer Society, 2009.

APPENDIX

A. PROOF OF FORWARD UNLINKABILITY OF USER LOG ENTRIES

THEOREM 4. *In the random oracle model, under the DDH assumption, the proposed logging scheme provides computational forward unlinkability of user log entries, according to Definition 4.*

PROOF. We use the game based proof technique proposed by Shoup [20]. The initial game corresponds to the proposed scheme. Only the differences with the previous game are explicitly mentioned.

- **Game 0:**
  - `CreateUser(`λ`)` creates a new user U_i and logs the first data λ for that user as follows:
    - $\mathrm{U}_i \leftarrow \mathrm{PK} \leftarrow$ `GenerateKey()`
    - $AK_0 \leftarrow$ `Rand`$(|H(\cdot)|)$
    - `State`$(\mathrm{ID}_{\mathrm{U}_i}, IC) \leftarrow \mathtt{MAC}_{AK_0}\left(\mathrm{ID}_{\mathrm{U}_i}\right)$
    - `State`$(\mathrm{ID}_{\mathrm{U}_i}, DC) \leftarrow$ null
    - `State`$(\mathrm{ID}_{\mathrm{U}_i}, AK) \leftarrow H(AK_0)$
    - `DrawUser(`U_i`,`U_i`)` $\rightarrow vuser$
    - `CreateEntry(`$vuser$`,`λ`)` $\rightarrow l$, return U_i, l.
  - `CreateEntry(`$vuser$`,`λ`)` creates a log entry l at log server L for data processor P and user U for given data λ as follows:
    - $(vuser, \mathrm{U}_i, \mathrm{U}_j) \leftarrow \mathcal{D}$, $\mathrm{U} = \mathrm{U}_i$ (if $b = 0$) or $\mathrm{U} = \mathrm{U}_j$ (if $b = 1$)
    - $d = \mathtt{Enc}_{\mathrm{PK_U}}\left(\lambda, \mathtt{Sign}_{\mathrm{SK_P}}(\lambda)\right)$
    - $IC_{\mathrm{U}} \leftarrow \mathtt{H}\left(\mathtt{State}(ID_{\mathrm{U}}, IC)\right)$
    - $DC_{\mathrm{U}} \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}}, AK)}\left(\mathtt{State}(ID_{\mathrm{U}}, DC), IC_{\mathrm{U}}, d\right)$
    - $IC_{\mathrm{P}} \leftarrow \mathtt{H}\left(\mathtt{State}(ID_{\mathrm{P}}, IC)\right)$
    - $DC_{\mathrm{P}} \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{P}}, AK)}\left(\mathtt{State}(ID_{\mathrm{P}}, DC), IC_{\mathrm{P}}, DC_{\mathrm{U}}, IC_{\mathrm{U}}, d\right)$
    - $\mathtt{State}(ID_{\mathrm{U}}, IC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}}, AK)}(IC_{\mathrm{U}})$
    - $\mathtt{State}(ID_{\mathrm{U}}, DC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{U}}, AK)}(DC_{\mathrm{U}})$
    - $\mathtt{State}(ID_{\mathrm{U}}, AK) \leftarrow \mathtt{H}\left(\mathtt{State}(ID_{\mathrm{U}}, AK)\right)$
    - $\mathtt{State}(ID_{\mathrm{P}}, IC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{P}}, AK)}(IC_{\mathrm{P}})$
    - $\mathtt{State}(ID_{\mathrm{P}}, DC) \leftarrow \mathtt{MAC}_{\mathtt{State}(ID_{\mathrm{P}}, AK)}(DC_{\mathrm{P}})$
    - $\mathtt{State}(ID_{\mathrm{P}}, AK) \leftarrow \mathtt{H}\left(\mathtt{State}(ID_{\mathrm{P}}, AK)\right)$
    - `Storage` $\leftarrow l \leftarrow (IC_{\mathrm{U}}, DC_{\mathrm{U}}, IC_{\mathrm{P}}, DC_{\mathrm{P}}, d)$
    - return l.
  - `CorruptLogServer()` returns `State`$(\mathrm{ID}_{\mathrm{P}}, AK)$, `State`$(\mathrm{ID}_{\mathrm{P}}, IC)$, `State`$(\mathrm{ID}_{\mathrm{P}}, DC)$ and $\forall\, \mathrm{U}_i$: `State`$(\mathrm{ID}_{\mathrm{U}_i}, AK)$, `State`$(\mathrm{ID}_{\mathrm{U}_i}, IC)$, `State`$(\mathrm{ID}_{\mathrm{U}_i}, DC)$.
  - Output of the experiment: $g \leftarrow \mathcal{A}$.

The event S_i is defined as $\mathcal{A}$ outputting a correct guess in game i, *i.e.* $g \stackrel{?}{=} b$.

$$Pr\left[S_0\right] = 1/2 \cdot \left(Pr\left[\mathbf{Exp}^0_{\mathcal{A}}(k) = 1\right] + Pr\left[\mathbf{Exp}^1_{\mathcal{A}}(k) = 1\right]\right)$$

- **Game 1:** The encryption of the data and signature on it is done under a random encryption key.
  - `CreateEntry(`$vuser$`,`λ`)`:
    - $d = \mathtt{Enc}_{\mathrm{PK}_R}\left(\lambda, \mathtt{Sign}_{\mathrm{SK_P}}(\lambda)\right)$ with $\mathrm{PK}_R \leftarrow$ `GenerateKey()`

The difference $|Pr\left[S_0\right] - Pr\left[S_1\right]|$ is the advantage of key privacy under chosen plaintext attack (IK-CPA) [3] of a distinguisher. For the used ECIES encryption scheme, this advantage is equal to the DDH advantage.

- **Game 2:** The random oracle $\mathcal{O}^{RO}(\cdot)$ is used to replace the outputs of the hash function by random values. The $\mathtt{MAC}_K(m)$ algorithm in our proposed scheme is instantiated with $\mathrm{HMAC}(K, m) = H\Big((K \oplus opad)||H\Big((K \oplus ipad)||m\Big)\Big)$.
  - `CreateUser(`λ`)`:
    - `State`$(\mathrm{ID}_{\mathrm{U}_i}, IC) \leftarrow \mathcal{O}^{RO}\Big((AK_0 \oplus opad)||\mathcal{O}^{RO}\Big((AK_0 \oplus ipad)||\mathrm{ID}_{\mathrm{U}_i}\Big)\Big)$
    - `State`$(\mathrm{ID}_{\mathrm{U}_i}, AK) \leftarrow \mathcal{O}^{RO}\Big(\mathtt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK)\Big)$
  - `CreateEntry(`$vuser$`,`λ`)`:
    - $IC_{\mathrm{U}} \leftarrow \mathcal{O}^{RO}\Big(\mathtt{State}(ID_{\mathrm{U}}, IC)\Big)$
    - $DC_{\mathrm{U}} \leftarrow \mathcal{O}^{RO}\Big((\mathtt{State}(ID_{\mathrm{U}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\Big)$
    - $IC_{\mathrm{P}} \leftarrow \mathcal{O}^{RO}\Big(\mathtt{State}(ID_{\mathrm{P}}, IC)\Big)$
    - $DC_{\mathrm{P}} \leftarrow \mathcal{O}^{RO}\Big((\mathtt{State}(ID_{\mathrm{P}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\Big)$
    - $\mathtt{State}(ID_{\mathrm{U}}, IC) \leftarrow \mathcal{O}^{RO}\Big((\mathtt{State}(ID_{\mathrm{U}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\Big)$
    - $\mathtt{State}(ID_{\mathrm{U}}, DC) \leftarrow \mathcal{O}^{RO}\Big((\mathtt{State}(ID_{\mathrm{U}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\Big)$
    - $\mathtt{State}(ID_{\mathrm{U}}, AK) \leftarrow \mathcal{O}^{RO}\Big(\mathtt{State}(\mathrm{ID}_{\mathrm{P}}, AK)\Big)$
    - $\mathtt{State}(ID_{\mathrm{P}}, IC) \leftarrow \mathcal{O}^{RO}\Big((\mathtt{State}(ID_{\mathrm{P}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\Big)$
    - $\mathtt{State}(ID_{\mathrm{P}}, DC) \leftarrow \mathcal{O}^{RO}\Big((\mathtt{State}(ID_{\mathrm{P}}, AK) \oplus opad)||\mathcal{O}^{RO}(\ldots)\Big)$
    - $\mathtt{State}(ID_{\mathrm{P}}, AK) \leftarrow \mathcal{O}^{RO}\Big(\mathtt{State}(\mathrm{ID}_{\mathrm{U}}, AK)\Big)$

Since we are in the random oracle model, $Pr\left[S_2\right] = Pr\left[S_1\right]$.

- **Game 3:** The state is only generated when the `CorruptLogServer` oracle is called. The IC and DC fields of the log entries are generated at random.
  - `CreateUser(`λ`)`:
    - $\mathrm{U}_i \leftarrow \mathrm{PK} \leftarrow$ `GenerateKey()`
    - `DrawUser(`U_i`,`U_i`)` $\rightarrow vuser$
    - `CreateEntry(`$vuser$`,`λ`)` $\rightarrow l$, return U_i, l.

– `CreateEntry(`$vuser$`,`λ`)`:
  * $d = \texttt{Enc}_{\mathrm{PK}_R}\Big(\lambda, \texttt{Sign}_{\mathrm{SK}_\mathrm{P}}(\lambda)\Big)$ with $\mathrm{PK}_R \leftarrow$ `GenerateKey()`
  * $IC_\mathrm{U} \leftarrow \texttt{Rand}(|H(\cdot)|)$
  * $DC_\mathrm{U} \leftarrow \texttt{Rand}(|H(\cdot)|)$
  * $IC_\mathrm{P} \leftarrow \texttt{Rand}(|H(\cdot)|)$
  * $DC_\mathrm{P} \leftarrow \texttt{Rand}(|H(\cdot)|)$
  * `Storage` $\leftarrow l \leftarrow (IC_\mathrm{U}, DC_\mathrm{U}, IC_\mathrm{P}, DC_\mathrm{P}, d)$, return l.

– `CorruptLogServer`:
  * $\forall\, \mathrm{U}_i$:
    · $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, IC) \leftarrow \texttt{Rand}(|H(\cdot)|)$
    · $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, DC) \leftarrow \texttt{Rand}(|H(\cdot)|)$
    · $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK) \leftarrow \texttt{Rand}(|H(\cdot)|)$
    · return $\texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, AK), \texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, IC), \texttt{State}(\mathrm{ID}_{\mathrm{U}_i}, DC)$
  * $\texttt{State}(\mathrm{ID}_\mathrm{P}, IC) \leftarrow \texttt{Rand}(|H(\cdot)|)$
  * $\texttt{State}(\mathrm{ID}_\mathrm{P}, DC) \leftarrow \texttt{Rand}(|H(\cdot)|)$
  * $\texttt{State}(\mathrm{ID}_\mathrm{P}, AK) \leftarrow \texttt{Rand}(|H(\cdot)|)$
  * return $\texttt{State}(\mathrm{ID}_\mathrm{P}, AK), \texttt{State}(\mathrm{ID}_\mathrm{P}, IC), \texttt{State}(\mathrm{ID}_\mathrm{P}, DC)$

In this game, the adversary has no possible way to detect any relation between log entries. Hence, the probability of success is equal to guessing $Pr[S_3] = 1/2$. For this step, the output of the random oracle needs to be uniformly distributed. Hence, the corresponding input need to be different from all previous ones. In each instantiation of the $\mathcal{O}^{RO}(\cdot)$, at least part of the input is truly random or the output of the random oracle. Furthermore, the adversary has no direct input to the random oracle, the data λ is put through a randomised encryption with a randomised public key. We obtain the following bound by using the birthday paradox:

$$|Pr[S_3] - Pr[S_2]| \leq 1 - \frac{(n-1)!}{(n-1-q)!(n-1)^q}$$

with $n = 2^{|H(\cdot)|}$ and q the number of queries made to the random oracle.

- **Conclusion:**

The following advantage for an adversary against future unlinkability of log entries is obtained for our scheme:

$$\begin{aligned}
\mathbf{Adv}^{FU}_{\mathcal{A}}(k) &= \frac{1}{2} \cdot \Big| Pr\Big[\mathbf{Exp}^{FU}_{\mathcal{A}}(k) = 1 | b = 0\Big] + \\
&\quad Pr\Big[\mathbf{Exp}^{FU}_{\mathcal{A}}(k) = 1 | b = 1\Big] - 1 \Big| \\
&= |Pr[S_0] - Pr[S_3]| \\
&= |Pr[S_0] - Pr[S_1] + Pr[S_2] - Pr[S_3]| \\
&\leq |Pr[S_0] - Pr[S_1]| + |Pr[S_2] - Pr[S_3]| \\
&\leq \mathbf{Adv}^{DDH}(k) + 1 - \frac{(n-1)!}{(n-1-q)!(n-1)^q} .
\end{aligned}$$

In the random oracle model, under the DDH assumption, this advantage is negligible. □

B. PROOF OF UNLINKABILITY OF USER IDENTIFIERS

THEOREM 5. *Under the DDH assumption, the proposed logging scheme provides computational forward unlinkability of user identifiers, according to Definition 5.*

PROOF. First, we will show that the distribution of the randomised user identifiers is equal to the distribution of user identifiers, uniformly at random. User identifiers are generated by `GenerateKey()`$\rightarrow \mathrm{SK}, \mathrm{PK}$, where $\mathrm{SK} = x$ is chosen uniformly at random modulo n and $\mathrm{PK} = X = xG$, for G a generator of the underlying curve. A blinded key is generated as follows: $X' = X + rG = (x + r)G$, with r uniformly chosen at random modulo n. The distribution of $(x + r)$ is also uniformly at random modulo n, since x and r are independent.

Second, we will show by reduction that there exists no adversary against the unlinkability of user identifiers. Assume an adversary $\mathcal{A}$ with non-negligible probability of success in determining whether or not $\langle \mathrm{ID}_\mathrm{U}, \mathrm{ID}'_\mathrm{U}, d \leftarrow l \rangle$ is a valid tuple, *i.e.* of the form $\langle x_1 G, x_2 G, \texttt{Enc}_{X_1}\Big(M_f, x_2 - x_1, \dots\Big)\rangle$. The used encryption scheme is ECIES, for which the ciphertext (R, c, m) of a plaintext p under public key X is computed as follows:

$$R = rG \quad c = \mathrm{ENC}_{K_0}(p) \quad m = \texttt{MAC}_{K_1}(c), \text{ with } r \in_R \mathbb{Z}_n \text{ and}$$
$$K_0 || K_1 = KDF\Big(xcoord(rX)\Big) .$$

Given this adversary $\mathcal{A}$, another adversary can be constructed against an instance of the DDH problem: given $\langle A = aG, B = bG, C \rangle$, determine whether or not $C = abG$. For a valid DDH tuple, $\mathcal{A}$ will have a non-negligible advantage when given the following: for $t \in_R \mathbb{Z}_n$, $\langle A, A + tG, \texttt{Enc}_A\Big(M_f, t, \dots\Big)\rangle$, where for the encryption we use $rG = B$ and $K_0 || K_1 = KDF\Big(xcoord(C)\Big)$. If the DDH tuple was not valid, $\mathcal{A}$ can only guess and has probability of $1/2$ to win the game. □